Ссылка на статью:
// Ученые записки УлГУ. Сер. Математика и информационные технологии. 2022, № 2, с. 38-48.

Поступила: 12.11.2022
Окончательный вариант: 12.11.2022

УДК 519.7

Высокоскоростная программная реализация алгоритмов шифрования из ГОСТ Р 34.12-2015

Гафуров И. Р.

gafurov.ils@yandex.ru
УлГУ, Ульяновск, Россия

В работе проводится высокоскоростная программная реализация алгоритмов «Магма» и «Кузнечик», используя процессорные инструкции SSE2, AVX, AVX2 и LUT-таблицы. Применение данных процессорных инструкций позволило увеличить скорость шифрования по сравнению с уже известными реализациями на примере OpenSSL.

Ключевые слова: *ГОСТ Р 34.12-2015, технология SIMD, язык ассемблера MASM, LUT-таблицы.*

Введение

В современном мире наблюдается активное применение криптографических методов в информационных системах. Однако увеличение компьютерных сетей и объёма данных, передаваемых между ними, требует высокой пропускной способности каналов передачи данных и высокой скорости шифрования данных, чтобы третьи лица не смогли завладеть информацией личного, государственного или военного характера. Поэтому вопрос оптимизации программ, в том числе криптографических, является актуальным.

Актуальность подкрепляется и теми фактами, что ни один компилятор не способен полностью проанализировать компилируемый код и произвести некоторые оптимизации, он не обладает базой алгоритмов, да и тем, что плохо написанную программу не сможет оптимизировать ни один компилятор. Человек, знающий язык ассемблер, взглянет на дизассемблированный код программы и увидит избыточность команд. Он сможет написать более оптимизированный вариант программы, чем её создаст компилятор.

Важно подметить, что написание оптимизированного варианта программы является трудоёмким процессом, занимающим немало времени. Это обусловлено тем, что каждый

алгоритм обладает своими особенностями и нуждается в индивидуальном рассмотрении. Проведём оптимизацию, используя принцип SIMD (Single Instruction, Multiple Data) и построение LUT-таблиц (Look Up Table).

1. Программная реализация алгоритма «Кузнечик» с использованием предвычисленных таблиц

Из [1] известно, что функция шифрования $E(a)$ и расшифрования $D(a)$, где $a, y \in V_{128}$, представляются в следующем виде:

$$E_{k_1,\dots,k_{10}}(a) = X[k_{10}]LSX[k_9] \dots LSX[k_2]LSX[k_1](a)$$

$$D_{k_1,\dots,k_{10}}(a) = X[k_1]S^{-1}L^{-1}X[k_2] \dots S^{-1}L^{-1}X[k_9]S^{-1}L^{-1}X[k_{10}](a)$$

Самые быстрые программные реализации обычно достигаются путем представления наиболее ресурсоёмких операций с помощью LUT-таблиц, при условии, что эти таблицы требуют разумного объема памяти. Наиболее ресурсоёмкими операциями в «Кузнечике» являются L-, LS-, $S^{-1}L^{-1}$-преобразование. Составим LUT-таблицы, упрощающие эти преобразования.

Начнём с L-преобразования. Для его выполнения необходима функция умножения чисел в поле Галуа над неприводимым полиномом $x^8 + x^7 + x^6 + x + 1$. Реализовать данную функцию можно, например, следующим образом:

```
uint8_t GF_256(uint8_t a, uint8_t b) {
    uint8_t result = 0;
    while (a && b) {
        if (b & 1) result ^= a;
        a = (a << 1) ^ (a & 0x80 ? 0xC3 : 0x00);
        b >>= 1;
    }
    return result;
}
```

Данная функция в преобразовании и составлении LUT-таб таблицы будет вызываться часто, поэтому составим такую предвычисленную таблицу Mul размерности 256×256, что $Mul_{i,j} = GF\_256(i, j)$, $Mul_{i,j} \in V_8$ и $i, j \in [0, 255]$. Получится следующая таблица с шестнадцатеричным представлением значений:

$$Mul = \begin{pmatrix} 00 & 00 & \dots & 00 & 00 \\ 00 & 01 & \dots & fe & ff \\ 00 & 02 & \dots & 3f & 3d \\ \dots & \dots & \dots & \dots & \dots \\ 00 & ff & \dots & f9 & 06 \end{pmatrix}_{256\times256}$$

L-преобразование можно осуществить путём шестнадцатикратного применения LFSR (linear feedback shift register). Введём матрицу C, первую строку которой составляет вектор констант из подпункта 4.1.2 в [1].

$$C = \begin{pmatrix} 94 & 20 & 85 & \dots & 20 & 94 & 01 \\ 01 & 00 & 00 & \dots & 00 & 00 & 00 \\ 00 & 01 & 00 & \dots & 00 & 00 & 00 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 00 & 00 & 00 & \dots & 01 & 00 & 00 \\ 00 & 00 & 00 & \dots & 00 & 01 & 00 \end{pmatrix}_{16\times16}$$

Пусть $Z = C^{16}$. Представим результат применения шестнадцати раз LFSR в следующем виде:

$$L(a) = aZ = (a_{15}, a_{14}, \dots, a_0)_{1\times16} \begin{pmatrix} 94 & 20 & 85 & \dots & 20 & 94 & 01 \\ 01 & 00 & 00 & \dots & 00 & 00 & 00 \\ 00 & 01 & 00 & \dots & 00 & 00 & 00 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 00 & 00 & 00 & \dots & 01 & 00 & 00 \\ 00 & 00 & 00 & \dots & 00 & 01 & 00 \end{pmatrix}^{16}_{16\times16}$$

Здесь вектор a – набор из 16 шифруемых байт, где $a_{i,} \in V_8$, $i \in [0, 15]$.

Свяжем с каждой строкой матрицы Z некоторое линейное отображение $L_i: V_8 \to V_{128}$, такое что $\forall b \in V_8: L_i(b) = b * Z_{i,15} \parallel b * Z_{i,14} \parallel \cdots \parallel b * Z_{i,0}$. Здесь знак $\parallel$ означает конкатенацию. Стоит отметить, что строки таблицы Z нумеруются не сверху вниз, а снизу вверх.

С учётом введённой нами ранее таблицы Mul, мы можем заменить произведение на обращение к этой таблице, т.е.

$$L_i(b) = Mul_{b,Z_{i,15}} \parallel Mul_{b,Z_{i,14}} \parallel \cdots \parallel Mul_{b,Z_{i,0}}.$$

Тогда для общего случая получим следующую функцию:

$$L(a) = L_{15}(a_{15}) \oplus L_{14}(a_{14}) \oplus \dots \oplus L_0(a_0).$$

Наша LUT-таблица имеет объём в 2^{16} байт. В коде программы её можно представить в виде матрицы размера 16×256, элементами которого являются 16-байтовые значения. Мы будем представлять в виде вектора длиной 2^{16}, элементами которого являются байты.

Теперь объединим L- и S-преобразования. Определим отображение $L_i^*: V_8 \to V_{128}$, $i \in [0,15]$, такое что

$$\forall b \in V_8: L_i^*(b) = Mul_{S(b),Z_{i,15}} \parallel Mul_{S(b),Z_{i,14}} \parallel \cdots \parallel Mul_{S(b),Z_{i,0}},$$

где $S(b)$ – нелинейное биективное преобразование. Следовательно, функция для общего случая будет иметь следующий вид:

$$LS(a) = L_{15}^*(a_{15}) \oplus L_{14}^*(a_{14}) \oplus \dots L_0^*(a_0).$$

Стоит обратить внимание, что элементы всех таблиц не зависят от других данных, подлежащих преобразованию, поэтому они могут быть рассчитаны заранее.

Объединим S^{-1}- и L^{-1}-преобразование. Обратное преобразование $S^{-1}L^{-1}$, необходимое для расшифровки, не может быть реализовано только с помощью LUT-таблиц. Причиной этому служит предшествование L^{-1}- над S^{-1}-преобразованием. Данная проблема была решена в [6]. Для решения проблемы представим функцию расшифрования иным образом.

Из подпункта 4.4.2 в [1] следует, что

$$D(a) = X[k_1]S^{-1}L^{-1}X[k_2] \dots S^{-1}L^{-1}X[k_9]S^{-1}L^{-1}X[k_{10}](a)$$

Следовательно,

$$D(a) = X[k_1]S^{-1}L^{-1}X[k_2] \dots S^{-1}L^{-1}X[k_9]S^{-1}L^{-1}X[k_{10}]S^{-1}S(a)$$

Т.к. L^{-1} – линейное преобразование, то

$$L^{-1}X[k_i]S^{-1}(a) = L^{-1}X[k_i](S^{-1}(a)) = L^{-1}(S^{-1}(a) \oplus k_i) =$$

$$= L^{-1}(S^{-1}(a)) \oplus L^{-1}(k_i) = L^{-1}S^{-1}(a) \oplus L^{-1}(k_i) = X[L^{-1}(k_i)]L^{-1}S^{-1}(a).$$

Из последних двух функций следует, что

$$D(a) = X[k_1]S^{-1}(L^{-1}X[k_2]S^{-1}) \dots (L^{-1}X[k_{10}]S^{-1})S(a) =$$

$$= X[k_1]S^{-1}(X[L^{-1}(k_2)]L^{-1}S^{-1}) \dots (X[L^{-1}(k_{10})]L^{-1}S^{-1})S(a)$$

Теперь мы поменяли местами преобразования S^{-1} и L^{-1}. Преобразование $L^{-1}S^{-1}$ может быть реализовано как и LS по LUT-таблицам L_i^{**}:

$$L^{-1}S^{-1}(a) = L_{15}^{**}(a_{15}) \oplus L_{14}^{**}(a_{14}) \oplus \dots L_0^{**}(a_0).$$

2. Программная реализация алгоритма «Кузнечик» с использованием векторных инструкций SSE2

Рассмотрим 128-битные xmm регистры, в которые хорошо помещаются шифруемые блоки, и следующие инструкции SSE2:

1. movdqu – перемещение не выровненных упакованных целочисленных значений;
2. movdqa – перемещение выровненных упакованных целочисленных значений;
3. pxor, pand, pandn – побитовое, исключающее ИЛИ, И, И НЕ (соответственно);
4. psrlw, psllw – логический сдвиг вправо и влево (соответственно) упакованных слов;
5. pextrw – извлечение указанного слова из регистра и перемещение его в другой регистр.

Первые две инструкции будут применяться для загрузки и выгрузки данных в регистры, инструкции с третьего пункта для булевых операций, с четвёртого пункта для побитовых сдвигов, с пятого для выгрузки байт регистра. Более подробную информацию об этих инструкциях можно узнать в [7].

Как говорилось в пункте 1, наша предвычисленная таблица LS-преобразования является вектором, содержащим 2^{16} байт или 4096 блоков по 16 байт. Выходит, что блоки в этом векторе хранятся друг за другом и адрес первого байта очередного слагаемого вычисляется по формуле:

$$\text{Address}(i, j) = \text{rax} + 4096i + 16j,$$

где rax – базовое смещение, i – индекс строки матрицы (номер блока), j – индекс байта очередного слагаемого. Из-за того, что блоки хранятся друг за другом, необходимо умножать j на 16 (использовать побитовый сдвиг влево на 4), чтобы получить смещение.

Перед тем как приступить к реализации основного цикла алгоритма «Кузнечик», вычислим заранее смещения с применением xmm регистров.

Выделим из 16-ти байтного блока $a = a_{15} \| a_{14} \| \cdots \| a_0$, $a_{0,1,\dots,15} \in V_8$ чётные и нечётные байты. Для этого введём вектор $\mathrm{Mask} = (00, \mathrm{FF}, 00, \dots, \mathrm{FF})_{1\times 16}$.

Данный вектор послужит маской для выделения необходимых байт с помощью инструкций pand и pandn. Напишем на языке ассемблера MASM предвычисление смещений.

Пусть в регистр xmm0 загружается шифруемый блок a, в xmm1 и xmm2 – маска, [rdx] – раундовый ключ. Тогда предвычисление смещений будет выглядеть следующим образом:

```
pxor xmm0, [rdx]
pand xmm2, xmm0
pandn xmm1, xmm0
psllw xmm1, 4
psrlw xmm2, 4
```

Таким образом, вычисление адресов слагаемых производится за две инструкции:

```
pextrw eax, xmm1, 0h; извлечение 16-битного слова из xmm-регистра
movdqa xmm0, [r11 + rax + 00000h]; применение LUT-таблицы
```

где r11 – базовое смещение, 00000h – строка LUT-таблицы, rax – i-ое смещение (в данном случае i=0h).

Перейдём к реализации раунда шифрования. Начнём с инициализации регистров.

```
lea r8, [tableLS+01000h]; загружаем адрес LUT-таблицы со смещением
lea r11, [tableLS]; загружаем адрес LUT-таблицы без смещения
movdqa xmm4, [mask]; загружаем в регистр маску выделения байт
movdqu xmm0, [rcx]; загружаем в регистр шифруемый блок по адресу первого принимаемой функцией аргумента
```

Перейдём к реализации раунда шифрования.

```
pxor xmm0, [rdx]; сложение шифруемого блока и раундового ключа (адрес второго аргумента функции)
pand xmm2, xmm0; загрузка в регистры xmm2 и xmm1 чётных и нечётных байт
pandn xmm1, xmm0;
psllw xmm1,4; приведение к используемым в адресной арифметике значениям
psrlw xmm2, 4
```

```
   pextrw eax, xmm1, 0h; извлечение 16-битного слова из xmm -
регистра
   movdqa xmm0, [r11 + rax + 00000h]; применение LUT-таблицы
   …
   pextrw eax, xmm2,7h;
   pxor xmm3, [r8 + rax + 0e000h]
   pxor xmm0, xmm3;
   pxor xmm0, [rdx];
```

Зашифрованный блок в результате будет записан в регистр xmm0.

3. Программная реализация алгоритма «Кузнечик» с использованием векторных инструкций AVX2

Теперь рассмотрим 256-битные ymm регистры, в которые хорошо помещаются два шифруемых блока. В данном случае одно LSX-преобразование применяется сразу на два блока.

Как и в случае использования инструкций SSE2, вычислим заранее смещения с применением ymm регистров. Из 32-ух байтного блока (содержит два блока по 16 байт) $a = a^* \| a^{**} = a^*_{31} \| \cdots \| a^{**}_{15} \| \cdots \| a^{**}_0$, $a_{0,1,\ldots,31} \in V_8$ также выделяются чётные и нечётные байты. Вектор, содержащий маску, примет следующий вид:

$$\text{Mask} = (00, \text{FF}, 00, \ldots, \text{FF})_{1\times 32}.$$

Принцип выделения тот же, что и в предыдущем пункте, за исключением того, что теперь применяются инструкции vpand, vpandn, vpsllq и vpsrlq.

Реализация раунда шифрования от предыдущего пункта отличается лишь загрузкой шифруемых блоков в ymm регистры и, естественно, использованием аналогов инструкций AVX2. Описание различных инструкций процессора можно посмотреть в [3].

4. Программная реализация алгоритма «Магма» с использованием инструкции AVX

В перечень инструкций AVX входит инструкция VPSHUFB [4], копирующая байты из регистра-источника в регистр-приемник по правилу, описанном в индексном регистре. Работу данной инструкции можно увидеть на рис. 1.

В одном xmm-регистре умещается два S-бокса, следовательно, все блоки замены можно уместить в четырёх xmm-регистрах. Однако, тетрады байта необходимо помещать в разные регистры, при чём младшие тетрады помещаются в младшие тетрады байты xmm-регистра, а старшие – в старшие тетрады. На рис. 2 представлено размещение, которое будет применяться при программной реализации алгоритма шифрования. Данное размещение обусловлено применением AVX инструкции VPSHUFB.

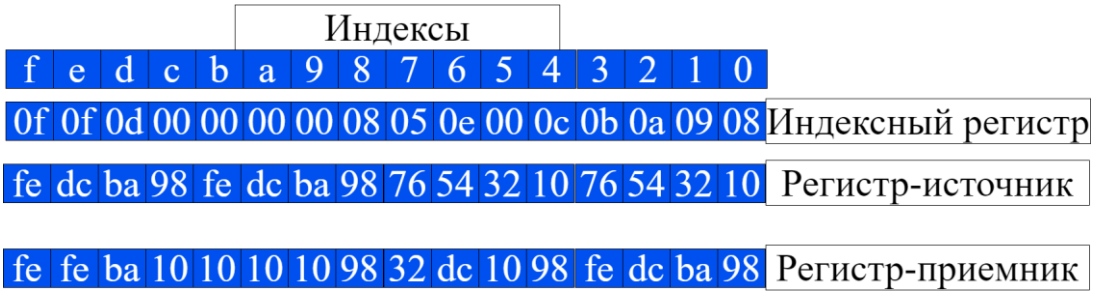

Рис. 1. Копирование байт в регистр-приемник из регистра-источника с помощью инструкции VPSHUFB

Таблицы замены для тетрад (регистры источники)	21	bf	c3	90	67	ad	f8	4e	39	8d	55	0a	d2	e6	74	1c
	00	e6	39	4c	14	87	71	be	ad	ca	2f	92	68	f5	d3	5b
	bc	92	e4	3b	5e	a3	09	70	6d	f6	41	d8	1a	25	8f	c7
	f7	03	da	bd	70	4b	e4	1f	cc	51	a9	96	35	22	8e	68

Рис. 2. Размещение узлов замены на xmm-регистрах

Нам также понадобится маска для выделения необходимых тетрад. В нашем случае она будет задана в следующем виде:

```
extern "C" uint8_t mask[6][16] =
{
  {0x00, 0x00, 0x0f, 0x0f, 0x00, 0x00, 0x0f, 0x0f, 0x00, 0x00,
0x0f, 0x0f, 0x00, 0x00, 0x0f, 0x0f},
  {0x0f, 0x0f, 0x00, 0x00, 0x0f, 0x0f, 0x00, 0x00, 0x0f, 0x0f,
0x00, 0x00, 0x0f, 0x0f, 0x00, 0x00},
  {0xf0, 0x00, 0x00, 0x0f, 0xf0, 0x00, 0x00, 0x0f, 0xf0, 0x00,
0x00, 0x0f, 0xf0, 0x00, 0x00, 0x0f},
  {0x00 ,0xf0, 0x0f, 0x00, 0x00 ,0xf0, 0x0f, 0x00, 0x00 ,0xf0,
0x0f, 0x00, 0x00 ,0xf0, 0x0f, 0x00},
  {0x00, 0x0f, 0xf0, 0x00, 0x00, 0x0f, 0xf0, 0x00, 0x00, 0x0f,
0xf0, 0x00, 0x00, 0x0f, 0xf0, 0x00},
  {0x0f, 0x00, 0x00, 0xf0, 0x0f, 0x00, 0x00, 0xf0, 0x0f, 0x00,
0x00, 0xf0, 0x0f, 0x00, 0x00, 0xf0}
};
```

Перейдём к программной реализации. Мы задействуем следующие регистры: rcx – хранит адрес левой половины шифруемых блоков; rdx – хранит адрес правой половины шифруемых блоков; rax – зашифрованный блок; xmm2 – индексы младших тетрад 1,2 байта и индексы старших тетрад 3,4 байта (индексный регистр); xmm3 - индексы младших тетрад 3,4 байта и индексы старших тетрад 1,2 байта (индексный регистр); xmm6 – левая часть шифруемых блоков; xmm7 – правая часть шифруемых блоков; xmm0, xmm1,

xmm12, xmm13 – узел замены; xmm10, xmm11 – для хранения результатов промежуточных вычислений; xmm15 – итерационный ключ.

В первую очередь инициализируем узлы замены:

```
movdqa xmm0, [S]
movdqa xmm1, [S+10h]
movdqa xmm12, [S+20h]
movdqa xmm13, [S+30h]
```

Здесь S – матрица, строки которой являются блоками замены, полученными по рис. 2. Реализуем один раунд шифрования:

```
movdqa xmm7, xmmword ptr [rcx]; загружаем левую половину шифруемого блока
movdqa xmm6, xmmword ptr [rdx]; загружаем правую половину шифруемого блока
movdqa xmm15, xmmword ptr [key]; загружаем итерационный ключ
paddd xmm7, xmm15; складываем часть блока с итерационным ключом
vpand xmm2, xmm7, xmmword ptr [mask]; готовим блок замен для 1 и 2 байт
vpsrlq xmm3, xmm7, 4
vpand xmm11, xmm3, xmmword ptr [mask+10h]
por xmm2, xmm11
; готовим блок замен для 3 и 4 байт
…
vpshufb xmm10, xmm13, xmm2; блок замен для 1 и 2 байт
pand xmm10, xmmword ptr [mask+20h]
vpshufb xmm11, xmm12, xmm2
pand xmm11, xmmword ptr [mask+30h]
; блок замен для 3 и 4 байт
…
pxor xmm6, xmm7
; циклический сдвиг на 11 бит влево
vpslld xmm10, xmm11, 11; побитовый сдвиг на 11 бит влево
psrld xmm11, 21; побитовый сдвиг на 21 бит вправо
pxor xmm6, xmm10
pxor xmm6, xmm11
```

После завершения алгоритма остаётся вернуть зашифрованный блок.

5. Сравнение полученных результатов

Возможности целевой платформы очень важны для реализации и выполнения программы. Поэтому перед переходом к сравнению полученных результатов опишем конфигурацию ЭВМ и среду разработки, которые применялись при реализации.

Замеры проводились на персональном компьютере со следующей конфигурацией:

1. **CPU:** Intel Core i3 9100f
2. **RAM:** Kingston 2x8 ГБ, работающие на частоте 2400 МГц
3. **SSD:** Kingston A400 240 ГБ

Программная реализация была произведена с применением языка ассемблера MASM и Си в среде разработки Visual Studio 2022. Операционная система: Windows 10 версии 22H2.

Скорость шифрования программной реализации алгоритма «Магма» будет сравниваться с реализацией из статьи [3] и библиотекой OpenSSL. При шифровании данных, находящихся на SSD, был применён режим шифрования ECB из [2].

Таблица 1. Скорости шифрования файлов разными программными реализациями алгоритма «Магма»

	Алгоритмическая оптимизация из [3]	Ассемблерная вставка из [3]	OpenSSL	Реализация с использованием инструкций AVX
Скорость шифрования	60 Мбайт/с	85 Мбайт/с	55 Мбайт/с	200 Мбайт/с

По таблице видно, что применение AVX инструкций способствует значительному увеличению скорости шифрования. Важно подметить, что полученный показатель не является пределом, ведь к «Магме», как и к «Кузнечику» можно применить инструкции AVX2 с ymm регистрами. А с увеличением распространённости процессоров с AVX512 станут доступны zmm регистры, которые в два раза больше, чем ymm, т.е. имеют размер в 512 бит.

Скорость шифрования программной реализации алгоритма «Кузнечик» будет сравниваться с реализацией в [5]. При шифровании данных, находящихся на SSD был применён режим шифрования ECB из [2].

Таблица 2. Скорости шифрования файлов разными программными реализациями алгоритма «Кузнечик».

	Реализация с использованием инструкций SSE2	Реализация с использованием инструкций AVX2	Реализация «Кузнечика» из [5]
Скорость шифрования	195 Мбайт/с	190 Мбайт/с	30 Мбайт/с

По таблице видно, что скорость шифрования с применением инструкций AVX2 чуть меньше, чем с инструкциями SSE2. Причиной этому служит трата времени на загрузку в ymm регистры байты из xmm. Реализация алгоритма шифрования из [5] оказалась медленной по причине отсутствия в нём методов оптимизации. В целом, она послужила примером того, насколько умение оптимизировать код программы способно увеличить скорость его выполнения.

Заключение

В работе были реализованы алгоритмы шифрования из ГОСТ Р 34.12-2015 с применением процессорных инструкций и LUT-таблиц. Результаты проделанной работы показали эффективность выбранного подхода к решению вопроса оптимизации программы.

Продолжение решения вопроса высокоскоростной реализации криптографических алгоритмов в перспективе способствует написанию криптографической библиотеки, которая сможет как конкурировать с ходовыми (например, OpenSSL и Crypto++), так и стать «ядром» различных средств защиты (например, для криптоконтейнера). В ближайшем будущем реализуем данные алгоритмы шифрования с другими инструкциями процессора и технологиями распараллеливания (OpenCL, CUDA и т.п.), чтобы получить хорошую основу криптографической библиотеки, способной быстро шифровать/расшифровывать информацию на различных компьютерах «выжимая» из них все соки.

Список литературы

1. ГОСТ Р 34.12-2015. *Информационная технология. Криптографическая защита информации. Блочные шифры*. М.: Стандартинформ, 2016.
2. ГОСТ Р 34.13-2015. *Информационная технология. Криптографическая защита информации. Режимы работы блочных шифров*. М.: Стандартинформ, 2016.
3. Гафуров И.Р. Методы оптимизации программной реализации блочного шифра «Магма» // *Ученые записки УлГУ. Сер. Математика и информационные технологии. УлГУ.* Электрон. журн. 2022, № 1, с. 8-16.
4. Даниэль Куссвюрм. *Профессиональное программирование на ассемблере x64 с расширениями AVX, AVX2 и AVX-512* / пер. с анг. В. С. Яценкова. М.: ДМК Пресс, 2021. 628 с.
5. Дроботун Е. Извращения с импортозамещением. Работаем с алгоритмом блочного шифрования «Кузнечик» из ГОСТ 34.12-2015 // *Журнал «Хакер»*. 2017, № 216, с. 216-231.
6. Borodin M. A., Rybkin A. S. High-Speed Software Implementation of Kuznyetchik block cipher // *Information Security Problems. Computer Systems*. 2014, № 3, p. 67-73.
7. *Intel®64 and IA-32 Architectures Software Developer's Manual.* Intel Corporation, 2016.

High-speed software implementation of encryption algorithms from GOST R 34.12-2015

Gafurov, I. R.

gafurov.ils@yandex.ru

Ulyanovsk State University, Ulyanovsk, Russia

In this work, a high-speed software implementation of the Magma and Grasshopper algorithms is carried out using processor instructions SSE2, AVX, AVX2 and LUT tables. The use of these processor instructions made it possible to increase the encryption speed compared to already known implementations on the example of OpenSSL.

Keywords: *GOST R 34.12-2015, SIMD technology, MASM assembly language, LUT tables.*